NOνA UpMu Manual

Collated instructions and tips for the upward-going muon analysis: simulation, reconstruction, analysis, and general computing in NOνA

v0.05
Rob Mina
March 2016

Introduction

Over the course of working on this project in 2014 and 2015, I found that a great deal of my time has been spent learning how to use the necessary computing tools. Many of these tools, like SAM for users and the runs database, have limited or outdated documentation. Others, though they may be fairly well documented, are quite complex and therefore require a good bit of trial and error to use well. A third class of tools were modified or created specifically for this effort, and it is therefore the responsibility of those within this group to document them. The primary purpose of this document is to fill in the gaps of missing documentation and to provide a single source of information for all the various tools that have so far proved to be useful for this project.

An auxiliary purpose is to provide a record, as complete as possible, of relevant administrative information such as the names of the important datasets, their locations, and the particular parameters used in specifying them.

Part 1 is a general overview of the relevant parts of NOνA's software framework. This is not meant to be a replacement for the existing documentation, but should introduce all of the concepts and vocabulary that will be needed later. Part 1 will also include a description of each step in the process of creating the custom UpMuRecoAna ntuples used in the analysis.

Part 2 will consist of detailed descriptions and instructions for each of the computing tools used in the process, including SAM, SAM for users, jobsub, NOνA's grid submission script *submit_nova_art.py*, pnfs, and xrootd.

Part 3 is on general considerations when scripting, but it will also describe some of the scripts used to interface with these tools and the particular considerations taken when writing them.

Part 4 will focus on three tools specifically modified or created for the upward-going muon search: the atmospheric neutrino flux driver, WimpSim, and the UpMu analysis ntuples (incl. the modules used to create them).

Finally, Part 5 will detail each dataset created so far for use in the UpMu analysis.

There is also an extensive glossary briefly defining much of the jargon used in the text. Appendix D may be of particular interest too, as it contains a list of outstanding questions and problems in this project.

All instructions in this document will involve using SRT and gnumake to build NOvA code. The only context in which it is necessary to use cmake is when developing and testing NOvADDT code for use online.

Part 1: The NOvA software framework

The purpose of this section is to introduce vocabulary and concepts necessary to understand how files are processed and produced for the UpMu analysis, as well as to provide an extensive introduction to the basic techniques used.

This section and the rest of the document assumes a familiarity with a terminal environment and shells like Bash. If you have never used Linux or a terminal before, you should refer to Appendix A before continuing.

The best place to look for documentation on computing in NOvA is the Novaart Redmine wiki. Many of the links provided in this document are to pages on that site, and it should be the first place you look when you have a question about anything software or computing-related on NOvA. The data-driven trigger (DDT, described in a later section) group maintains a separate wiki page here.

Logging into the NOvA VMs

Almost all interactive work (*eg*. coding) involved in this or any NOvA analysis project is done on one of the NOvA general-purpose virtual machines (novagpvm{01..15}@fnal.gov). Logging into one of these nodes involves obtaining a valid kerberos ticket in the FNAL.GOV domain, and using ssh to open a tunnel to the virtual machine.

You must have openssh installed and configured properly to forward a kerberos ticket. From a fresh installation of Ubuntu (and most derivatives thereof), you can install the latest version of openssh by opening a terminal and executing this command:

```
sudo apt-get install openssh-client
```

You will be prompted to input your user password. Once this completes you must edit your ssh configuration file: `~/.ssh/config`

You may set a blanket rule that applies to every connection to the .fnal.gov domain by placing the following text into the config file:

```
Host *.fnal.gov
     ForwardAgent yes
     ForwardX11 yes
     ForwardX11Trusted yes
     GSSAPITrustDns yes
```

```
    GSSAPIKeyExchange yes
    StrictHostKeyChecking no
    UserKnownHostsFile /dev/null
    GSSAPIDelegateCredentials yes
```

I prefer to use separate rules for each of the 15 VMs, as in this case with VM number 03:

```
Host nova03
    User ram2aq
    HostName novagpvm03.fnal.gov
    ForwardAgent yes
    ForwardX11 yes
    ForwardX11Trusted yes
    GSSAPITrustDns yes
    GSSAPIKeyExchange yes
    StrictHostKeyChecking no
    UserKnownHostsFile /dev/null
    GSSAPIDelegateCredentials yes
```

While in the former case I would have to type `ssh ram2aq@novagpvm03.fnal.gov` to connect, with the latter rule I could use just `ssh nova03.`

Before you can log into one of the nodes, you must have a valid Kerberos ticket. Kerberos is the authentication technology used by Fermilab to ensure that only authorized users may connect to its computing resources. You can read a little about how to set it up on this page. Unfortunately the link to the current krb5.conf file given on that page **is no longer valid**, and therefore the step-by-step instructions at the bottom of the page **won't work**. Here are the steps you should follow to install kerberos and obtain the correct configuration.

- `sudo apt-get install krb5-user`
    - you can mostly ignore the prompts, but you should use FNAL.GOV when prompted to specify the domain to avoid having to type it later when doing kinit.
- `wget http://computing.fnal.gov/authentication/krb5conf/Linux/krb5.conf`
    - replace Linux with OSX if you're using a Mac
- `sudo mv krb5.conf /etc/`

When you are ready to connect to one of the nodes, use:

- `kinit <your Fermilab username>`

    - enter your Fermilab kerberos password
- `ssh nova03`
    - or `ssh <your Fermilab username>@novagpvm03.fnal.gov`

If you see an error message, try doing `kinit -A <your Fermilab username>`. This will get an addressless ticket, which is necessary when connecting from behind a NAT.

Writing a setup_nova function

The first time you log into one of the nodes, you should configure your setup_nova function as described on this page. It indicates that you can edit **either** your ~/.bashrc **or** ~/.profile files, which is true for interactive shells. However, if you ever decide to use screen[1], you will want to have the function defined in **both**. Personally, I prefer using the ~/.bash_profile, rather than the ~/.profile file.

As another note, the previously linked page tells you how to source a particular NOvASoft release by passing the -r flag, but you should also know that each release contains multiple builds that are specified using the -b flag. By default, the debug build is setup. This build is useful because it compiles the code in such a way that a debugger can easily be used with the compiled libraries. However, it is much slower than the optimized code. To source the fastest available build, you should pass `-b maxopt` to the setup_nova function. Do this whenever you want to run a large job or when you are trying to test the speed of execution of your code.

Note that the `setup_nova` function is to be called each time you log into the node if you will be using SRT to build your NOvA code. There is another build system in NOvA that uses cmake, but this will not be covered here.

If you will be submitting grid jobs, you should add the following line to your setup_nova function:

```
kx509
```

This will obtain the necessary ticket to allow job submission and output file copying. If you will be using xrootd, add the following line to your setup_nova function:

```
voms-proxy-init --rfc --voms=fermilab:/fermilab/nova/Role=Analysis
--noregen
```

[1] screen is an immensely useful utility that allows the creating of multiple shells on the same host. Each can be detached, re-attached, and ended individually. Whatever command is running when the shell is detached will continue to run to completion or exception. Thus you can start a script or code, logout, get some lunch, and come back to a finished job. This cannot be done with a standard login shell.

Finally, if you will be using the SAM for users tools, you should add this temporary workaround to your setup_nova function:

```
export SSL_CERT_DIR=/etc/grid-security/certificates
```

The DDT and the upward-going muon triggers

The Data-Driven Trigger (DDT) is a piece of software that runs on the computer cluster at the far detector site in Ash River. The purpose of the trigger is to perform basic reconstruction and analysis on the live data in order to select "interesting" events for storage. It is not feasible to store 100% of the data produced by the detector, but 100% live time can still be achieved with a filter that rejects events containing only ("uninteresting") typical or background processes. As of the time of this writing, the trigger achieves 100% live time in its current configuration, though this has not always been the case. The trigger must process data as it is produced by the detector and before the data is ever written to disk, so that trigger code must be written efficiently and with computational cost always in mind.

The DDT contains several trigger "streams" representing different execution paths that lead to filter (or "trigger") modules. Different modules that use the same information from the event will share the same trigger stream up to the point at which the required objects diverge. This means that no module will be run more than once per event in the same configuration, eliminating redundancy at the module level. As an example of one trigger stream, here are the modules currently running in the ddupmu ("through-going upward-going muon") stream[2]:

1. NovaDDTHitProducer - converts hit-level information from the DAQ into DDT-format DAQHit objects
2. SortByTDC - sorts DAQHit objects in the event by the time of the hit (in TDC units).
3. SingletonRejection - removes hits that are likely noise because they are isolated in both time and space.
4. TimeSlice - Combines hits that are close in time (from both views) into TimeSlices.
5. RemoveNoise - Removes TimeSlices that contain too few hits to be useful.
6. SpaceSlice - Takes the TimeSlices and further divides them into TimeSpaceSlices based on the plane numbers of hits in each TimeSlice.
7. RemoveSpatialNoise - Removes hits from each TimeSpaceSlice that have cell numbers indicating a large separation in X or Y from other hits in the slice. Also removes slices with too few hits to be useful.

[2] This information was taken from the DDTGlobalConfiguration fcl file defining the DDT configuration for physics runs at the far detector: https://cdcvs.fnal.gov/redmine/projects/novaddt/repository/entry/trunk/DDTGlobalConfigurations/DDTGlobalConfiguration-FD.fcl

8. RemoveOneDSlices - Removes slices that contain too few hits in one view or the other to be useful for 3D track reconstruction.
9. HoughTracker - Use the Hough Transform to reconstruct linear 2D tracks within each slice in each view. Each Track object contains information about the starting and stopping coordinates and the list of hits associated with the track.
10. Merge2DTracks - Combine 2D tracks by comparing the planes of the track extremes between the views to produce Track3D objects.
11. UpMuTrigger - Use timing information for hits along each track, along with a suite of cleanup cuts designed to select tracks for which the timing information is clear, to select upward-going muons. This module "triggers" on likely upward-going tracks, causing the event data to be stored permanently. Events with no tracks that seem upward-going are not triggered and will not be recorded unless one of the other triggers selects them.

Note that each of the modules in this stream runs with a particular configuration, as defined in the relevant FHiCL (.fcl) file. Many of them have multiple configurations that are used by the different trigger streams. For example, the UpMuTrigger module is the final filter module in both the ddupmu ("through-going") and the ddcontained ("contained") trigger streams. These two configurations are both defined in the UpMuTrigger.fcl file. The job configuration file[3] defines which module configuration is used for each stream, as described in the next section.

The first step in the UpMu analysis is the storage of ddupmu and ddcontained data files containing events that caused the "through-going" and "contained" triggers to fire, respectively. These files are stored in a raw format derived directly from the DAQ. All subsequent steps in the process are run offline, either on the grid or interactively, using the art framework.

Running an art job on the vm's

This section assumes that you are able to log into one of the NOvA general-purpose virtual machines (novagpvm??@fnal.gov). If you have trouble configuring kerberos and openssh, please refer to the "Logging into the NOvA VMs" section. You will also need to have a working setup_nova function, which you can read about in the "Writing a setup_nova function" section.

For the remainder of this document, it is assumed that the reader has been introduced to the art framework. Perusing the art workbook will be helpful to those who have not worked with art before.

[3] In this case, DDTGlobalConfiguration.fcl.

You can find a detailed writeup on running NOvA art jobs here. The following discussion will repeat much of the material from that page but should serve to introduce all the necessary details to follow the rest of the document.

The preceding section lists the steps used to run the ddupmu trigger online in the context of an explanation of the DDT streams, but this explanation is also an example of an art job configuration that will be useful for the current discussion. In general, an art job is composed of just four components:

- **a configuration fcl file.** This file defines the list of modules that will be run, along with all the variables that will be set to configure each of the modules. The Mu2e collaboration has an excellent writeup describing the FHiCL language here.
- **a source or input file.** For DDT jobs (as above), the source is the raw data (.raw) coming from the DAQ. For offline jobs, the source is generally an art-formatted (.root) file containing some number of events with associated art products. The source filename is sometimes defined in the job configuration file. For simulation jobs, there may be no source file.
- **an output file.** All art products created by any of the modules in the job will be placed into the output file as long as an output stream is defined in the configuration. The output filename must be specified either in the configuration file or by passing the -o flag at the command line if an output stream is defined. In some cases, it is not desired that the art products should be stored; this is the case, for example, when running the trigger online. It is also possible to run a service called the TFileService within the job, which outputs a ROOT file containing arbitrary ROOT objects as specified and populated in the modules themselves. This is often used to create histograms or flat ntuples that can be used interactively outside the art framework.
- **binary and runtime libraries.** For running DDT online (as above), the `ddtfilter` binary is used. For running offline jobs (as in almost all cases in this project), the `nova` binary is used. The runtime libraries are the compiled shared library (.so) files that contain the necessary art modules and services. In NOvA, code is compiled at the package level, meaning that if a particular module is used (for example, the NovaDDTHitProducer, which is in the DDTCore package), the library that will ultimately be linked is that corresponding to the repository package containing the source code for that module (libDDTCore.so, in this case).

A very simple example of an art job is to combine two .root files with identical art products for two different subruns into a single .root file with all the art products for both subruns. The NOvASoft repo already has a configuration file that can do this: Utilities/concat_files.fcl. Here is the text of the file for convenience (with a small typo fix):

```
# Usage: nova -c concat_files.fcl -o <outfile> <infiles>
```

```
source:
{
       module_type: RootInput
       maxEvents:   -1
}

outputs:
{
       # Don't specify a fileName. Require user to specify output file explicitly
       out:
       {
              module_type: RootOutput
       }
}

physics:
{
       stream:      [ out ]
       end_paths:   [ stream ]
}
```

Note that, in the out stream, there is no output filename specified. This means that the user *must* set the filename by passing the -o flag at the command line (if an output filename had been specified, the -o flag could be used to override it, but would not be necessary). Note also that the source files are not specified, so they must be listed at the command line.

You may also notice that the source definition contains a maxEvents parameter. By setting this to -1, nova will combine all events in the source files together. Setting it to some positive integer *x* will cause nova to stop after processing *x* events. In this case, it will only put *x* events into the combined file, regardless of the number of events in each source file. This parameter can be overridden by passing the -n flag at the command line.

Note that if multiple source files are passed to the nova binary, **every source file must contain the same set of art products.** This can become an issue when trying to merge a large dataset into a smaller number of files. Each file in the dataset should have been created using the same configuration *and* version of NOvASoft, as otherwise there may be different products in some of the files. In general, data files in NOvA are named in such a way that it should be clear which version (tag) was used to create them. This constraint applies universally to all nova jobs, not just to the concat_files.fcl configuration.

Here is a list of the most useful flags that can be passed to the nova binary to set or override FHiCL parameters:

-c	specify the configuration FHiCL file.
-o	specify the output filename.
-s	specify the source filename. Passing this flag is optional, as all filenames passed to the nova binary will be used as source files unless preceded by one of the other flags.
-S	specify the name of a text file containing a list of source filenames, one filename per line.
-n	specify the maximum number of events to process.
-T	if the TFileService[4] is used in any modules, specify the filename of the .root (histogram) file that will be created.
--no-output	turn off all outputs (useful for testing timing).

You can call `nova -h` after calling `setup_nova` to see a full list of available flags.

Process for creating UpMu analysis ntuples

Once a ddupmu or ddcontained raw file has been produced by the DAQ (and trigger), a job (DAQ2RawDigit/daq2rawdigitjob.fcl) is run to convert the DAQ format into an art format file. This is done on the grid and automatically, so that in general the user does not need to do it. This job produces artdaq format files that contain RawDigit objects encoding the hit-level information from the triggered events. One artdaq file is created per raw file (so there is one file of each type per trigger per subrun).

As an aside, you can see a list of all art products present in an art file using the eventdump.fcl job configuration:

```
nova -c eventdump.fcl <source filename> -n 1
```

You can also view the event in the event display, although much of the functionality will not be available since it depends on reconstructed objects that are not yet present in the file:

```
nova -c evd.fcl <source filename>
```

You can read more about the event display on this page, which is, unfortunately, outdated.

[4] The TFileService is an art service that allows modules to produce ROOT objects like histograms and trees that will be stored in a separate .root file from the art output file. The UpMuAna and UpMuRecoAna modules (among many other analysis modules in NOvA) both use this service to store the desired output, which is not in the form of art events.

Another possible source of artdaq files is simulation. The steps involved in producing a sample using either WimpSim or the atmospheric flux driver are detailed in Part 4. The final product of these steps is a file that contains both the RawDigits for a simulated event and a number of objects detailing the truth information that went into the simulation. This truth information is used by the UpMuRecoAna module if the *IsSim* parameter is set to true in the UpMuRecoAna.fcl file.

The next step, reconstruction, is the first that should conceivably be run interactively on the VMs [5]. This step requires that at least one artdaq file be obtained and moved to the /nova/ana/ area, or that xrootd be used. See the relevant discussion in Part 2. The reconstructed objects that are currently used in this analysis are CellHits, Tracks from KalmanTrackMerge, Vertices from FuzzyKVertex, and Michel electron clusters from MichelEFilter. So far we have used the Production/fcl/prod_reco_numi_job.fcl configuration from the tagged NOvASoft release S15-05-04b to produce the reconstructed sample. This creates files that contain these objects and several others.

The final step is to run the analysis module, UpMuRecoAna, on the reconstructed sample. This produces flat analysis ntuples that can be used to select candidates, make plots, etc. The UpMuRecoAna module, along with the ntuples that it produces, are detailed in Part 4.

[5] In practice it is much more practical to use the grid to reconstruct a sample of even a dozen subruns. See the discussion on running on the grid in Part 2.

Part 2: Instructions for using various computing tools

The purpose of this section is to provide an introduction to the various tools employed so far in this project. Links to relevant documentation will be provided wherever possible, and supplemental instructions given.

SAM

SAM ("Sequential data Access via Metadata") is a Fermilab-wide solution to the difficulty of making the multi-petabyte data/MC archive available to the various experimental analysis and production efforts. A number of helpful links on using SAM in NOvA are available here. If you have no experience with SAM, see this slightly out-dated but still very educational tutorial.

In the context of this project, there are two primary use-cases for SAM and the accompanying shell utility `samweb`:

Use-case 1: I know there is a data file (let's call it ddcontained_artdaq_1.root, although this file doesn't exist so you'll have to replace it in the following commands to get anything to work) that contains an interesting candidate event. Some methods to obtain a list of files matching a certain set of criteria (run number, date, trigger stream, file format, etc.) are described in the previously linked tutorial. I want to run my analysis module on this file and view it in the event display. I have two choices to get a hold of this file:

1. Use xrootd to run reconstruction on the file without copying it to bluarc (/nova/ana/ or /nova/app/). This will be described in a later section on xrootd. This is the preferred method if I will be running only a few times on the file (which is probably the case here, since I will run reconstruction just once on the file and then run my module as many times as I may need on the output).
2. Use `samweb locate-file` to find the path to the file, then use `ifdh cp` to copy it to my /nova/ana/ area. Or use `ifdh fetch` (with the filename only) to copy it to my /nova/ana/ area. This is the preferred method if I will be running many times on the file.
    a. `samweb locate-file -h # view options for the locate-file command`
    b. `samweb locate-file ddcontained_artdaq_1.root`
        - # something like this will be output: `enstore:/pnfs/nova/production/raw2root/S14-08-19/fardet/000185/18522/all(4749@vpl523)`

- "enstore" is the system used to interface with the Fermilab tape archive, implying that this file is in the tape-backed filesystem. This means the file may be available on dcache (a disk space allowing quicker access to files than the tape), or it may only be available on tape, in which case it will take a significant amount of time to access it.
- the path we want is `/pnfs/nova/production/raw2root/S14-08-19/fardet/000185/18522/all`

c. `ifdh cp /pnfs/nova/production/raw2root/S14-08-19/fardet/000185/18522/all/ddcontained_artdaq_1.root /nova/ana/users/ram2aq/`
   - This will copy the file into my /nova/ana/ area, where I can then run on it using whatever modules I would like from one of the gpvm's.

Use-case 2: I want to produce a large reconstructed sample of files for some analysis. Because there are many files that I need to process, I will be running on the grid. Here are the steps involved in running a grid job:

1. Define a SAM dataset including all of the files to run on. Methods to define custom datasets are described in the previously linked tutorial. In particular, you can use the `samweb define-dataset` command or the dataset definition gui here. You can then interact with the dataset you have created using other `samweb` commands. This page has many useful pointers and examples, although it may be slightly outdated.
2. Run a test on one of the gpvm's to determine how long you expect the job to take per file. This requires obtaining a copy of one or more files, as described in Use-case 1. Grid jobs should take at least 1 and no more than 6 hours per instance. You should set the number of instances based on your expectation of the running time per input file in order to satisfy this constraint.
3. Setup a directory in your /pnfs/nova/scratch area to store the output from your job.
4. Create a test configuration job file and submit a test job to make sure that the larger job will run successfully and produce the desired output.
5. Update the test configuration to a full job configuration and submit the full job.
6. Monitor the status of the job and, when it has completed, check that the output is as expected.

Steps 3 through 6 will be discussed in detail in a later section on using the grid. For step 2, it will be preferable in this situation to use xrootd, which will also be described in a later section.

SAM for Users

Over the course of an analysis project, it may be necessary to produce a large number of files. In the context of this project, ~53,000 reconstructed files were generated for each of the two trigger samples. In order to efficiently run the analysis module on these files on the grid, it is very convenient to declare them to SAM so that they can be used as part of a typical dataset. SAM for Users is a suite of tools that allows analysis users to easily create new SAM datasets from files that they have produced (as opposed to files that were produced by the DAQ or the production group).

The previously linked page is the primary piece of documentation for this tool; however, there are three pieces of missing information:

1. At the bottom of the page, it indicates that this error: oops: `SSL error: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed (_ssl.c:581)` can be fixed by executing this command: `export SSL_CERT_DIR=/etc/grid-security/certificates.` I've found that this command must *always* be executed before using any of the tools described on the documentation page. When I first tried to do setup fife_tools, there was an error that I didn't bother to resolve. I found that it isn't necessary to do this explicitly, but this step may resolve the previously mentioned error.
2. The `sam_add_dataset` command will prepend a 16-digit UUID to the beginning of each file. Unfortunately this is done before the certificate validation step, so that the command may fail *after* renaming the first file. If this occurs, you will have to manually rename the first file to remove the UUID. Additionally, when using the `sam_clone_dataset` command, it will not copy the directory structure of the dataset's location. Instead, it will create subdirectories with unique 4-character names within the specified target directory, each of which will contain one file. If the --twodeep flag is passed, it will create a sub-subdirectory with 4-character name within each of the subdirectories. Each sub-subdirectory will contain one file.
3. The sam_add_dataset command declares all files in the source directory or the source file list to SAM and gives them minimal metadata. The metadata that it provides is not sufficient to run locally or on the grid using the files. You can use `sam_modify_dataset_metadata` to add some of the required fields (file_type, data_tier, online.detector) for the entire dataset with a single command. Others fields (online.runnumber, online.subrun, runs) are specific to each file and are therefore best handled using a script. The script I used when handling the DDUpMu and DDContained samples is discussed in Part 3.

Generally, SAM for Users should be used when working with a large set of files that will need to be run on using the grid. It also simplifies the process of moving files from the /pnfs/nova/scratch/users/ area, which provides temporary storage for files of uncertain usefulness, to the /pnfs/nova/users/ area, which is tape-backed and should be used only for files that will definitely be useful in the future. An explanation of pnfs and the tools available to interact with it follows in a later section on running NOvA jobs with the grid.

jobsub and general grid job submission

The grid is a powerful resource that allows parallelization of computing tasks at the binary level (that is, it allows many simultaneous executions of a particular binary with different inputs, as opposed to parallelizing the work that occurs within a single execution). In the context of NOvA, this means that users can run art modules or scripts on hundreds or even thousands of subruns (files) at once.

Currently, there are few restrictions or safeguards in place to prevent users from submitting jobs to the grid that will consume an exorbitant amount of computing resources (cpu time and disk space). It is therefore the responsibility of grid users to carefully consider the resources their jobs will consume and the usefulness of the expected output. It is good practice for users to thoroughly test the jobs that they wish to submit before doing so. Testing should occur on one of the interactive gpvm's first, and for large jobs further testing should be performed by submitting an identical but trivially small job to the grid and checking that the output of this test job was as expected.

Fermilab maintains a large computing grid (fermigrid) onsite for lab users working on its various experiments. Currently, NOvA and mu2e use the greatest number of slots on this grid. NOvA users may also submit jobs to external grid sites that participate in the Open Science Grid.

In the context of NOvA, most grid jobs will be submitted using NOvA's self-maintained submission scripts `runNovaSAM.py` and `submit_nova_art.py,` which are the subject of the next section. It is also possible to write one's own scripts and submit them as jobs to the grid. This may be useful for tasks that require the movement of many files (eg. merging the output of a previous grid job). One should use the JobSub client to accomplish this. The previously linked page gives a good overview of the available functionality, but for clarity here is a list of commands used so far for this project:

`jobsub_submit`	This is the interface for submitting any script to the grid. For nova art

	jobs, the submit_nova_art.py script is the preferred interface.
jobsub_q	Query the status of the job queue. This returns a list of all currently active jobs satisfying the specified dimensions. For example, passing `-G nova` will print a list of all jobs from NOvA users (warning - there are usually thousands of active jobs). Passing `--user <your_username>` will print a list of all active jobs submitted by you. Passing `--jobid <your_jobid>` will print a list of all active processes with the specified jobid prefix.
jobsub_fetchlog	Fetch a compressed (.tgz) archive containing logs, stdout, and stderr from a job. This command requires that a particular jobid be specified. Passing the --list flag will cause a list of all jobids (along with the submission time and date) for which logs are available to be printed. As a general user, you will only see jobs that you submitted in the list. If you are unfamiliar with tar and the options needed to extract from a compressed archive, see this helpful page. The archive contains the following files: `<your_username>-<job_name>-<submission_date_time>.sh` - the automatically-generated script that was run to submit your job. `<your_username>-<job_name>-<submission_date_time>.sh_<submission_date_time>_<jobid>_wrap.sh` - the script that was run on each remote node. `<your_username>-<job_name>-<submission_date_time>.sh_<submission_date_time>_<jobid>.cmd` - a file containing the values of many of the parameters for your job. `<your_username>-<job_name>-<submission_date_time>.sh_<submission_date_time>_<jobid>.log` - the Condor log file containing information about the job submission process. Look in this file if you had a problem with the job submission itself. `<your_username>-<job_name>-<submission_date_time>.sh_<submission_date_time>_<jobid>_<instance_number>.out` - a copy of the stdout output from this instance of your job (many jobs have dozens or hundreds of instances running in parallel on different nodes). Look in this file if you had a problem with a particular instance of your job. `<your_username>-<job_name>-<submission_date_time>.sh_<submission_date_time>_<jobid>_<instance_number>.err` - a copy of the stderr output from this instance of your job. Look in this file if you had a problem with a particular instance of your job. Note that jobsub will only record the first 1 MB and the last 4 MB of

	output from any single instance of your job. If you need to retrieve output from a job, don't rely on stdout - you should write your job so as to produce the required output in a file that will be directly written to an output directory.
jobsub_hold	You can change a job to the "hold" state, which will prevent the grid from assigning additional node time to it. The job will remain on the queue. Note that jobs that are currently running will continue to run until their allocated time has ended. If there are multiple instances of a job, each will have a unique jobid - 2376776.<instance_number>@fifebatch2.fnal.gov In this case, you can put all instances on hold with a single command: `jobsub_hold --jobid 2376776@fifebatch2.fnal.gov`
jobsub_release	This is the inverse operation to putting a job on "hold." As with jobsub_hold, you can omit the instance number to release all instances of a job with a single command. When a problem with SAM had caused a job to be in a hanging state, issuing a hold and then a release may cause the job to continue normally.
jobsub_rm	This will remove a job from the queue. If a job is currently running when the remove command is issued, it will continue to run on the current node until the allocated time has ended. If it does not complete by that time, it will be cancelled without completing or writing output.

NOvA grid submission scripts

If you have not already, you should read the previous section on SAM and the preceding section on general grid job submission.

This page describes in detail the various NOvA-maintained tools for submitting jobs to run the nova binary on the grid. Especially helpful is the submit_nova_art.py script, which takes as input a configuration file (passed with the -f flag to the script), and automates the entire process of setting up a SAM project and submitting the job. All jobs run so far for this project were submitted using this script. By writing a configuration file and passing it with the -f flag, it is easy to re-run a job multiple times without having to remember all the configuration options.

The submit_nova_art.py script will check that any output directory you specified is group writable, and will fail with an error message if it is not. To make a directory group writable, simply do `chmod g+w <path to directory>`.

A warning that is missing from the page: if you run a job and specify a test release by passing the `--testrel` flag, **do not** recompile the libraries from that release while the job is running (that means don't do a `make` within the test release). Doing so will cause any instances that start while the compilation is happening to fail. Also, if you pass the `--maxopt` flag with the `--testrel` flag (this is advisable in almost every case when running on the grid), you must ensure that you have already built the maxopt version of that test release before submitting the job. As discussed in the previous section on writing a setup_nova function, you should pass `-b maxopt` to specify the maxopt build when calling `setup_nova`. The submit_nova_art.py script has checks in place to ensure that you have sourced the correct release before submitting your job, but it does not check that a library exists with the correct build. This error will cause your jobs to fail immediately after being assigned a node - all the overhead of submitting a job and storing its output will have been wasted.

Example configuration files used for running reconstruction and for generating UpMu analysis ntuples (described in Part 4 below) can be found at /nova/app/users/ram2aq/reco_config.txt and /nova/app/users/ram2aq/UpMuRecoAna_dev/upmuconfig.txt. Note that the former uses --outTier to write the art output files (the reconstructed events) while the latter uses --histTier to write the TFileService histogram files (the ntuples).

You should also read this page which gives helpful information about monitoring grid jobs and handling output that has been written to the /pnfs/ area. It is important to ensure that your jobs are completing successfully and using the collaboration's computing resources responsibly. As such, you should monitor your active jobs periodically while they are running.

The previously-linked page does not mention that because of the structure of the file system used on dcache (pnfs), you should avoid writing more than a hundred files into any single directory. Fortunately runNovaSAM.py (and, by extension, submit_nova_art.py) offers two options (--runDirs and --hashDirs) that automatically organize output files into subdirectories to limit the number of files written to any single output directory. You should also avoid doing `ls` and `find` commands in pnfs as much as possible, as these commands are expensive and can strain the system's resources, which are shared by all the intensity frontier experiments.

As a final note on using the grid, keep in mind that running on the grid is an inherently complicated process and is thus prone to unexpected and frequent issues. The fact that relatively few users make use of grid resources mean that problems are sometimes allowed to persist for

much longer than for more widely-used resources. Solutions and workarounds are often counterintuitive and known to only a few expert users.

I'll refrain from including an exhaustive list of issues that I have encountered while using the grid, as it is likely that the list would rapidly become outdated. Instead, I'll offer three general pieces of advice about debugging problems with grid jobs:

1. *Check the logs*. If a problem occurs, grab the job output using `jobsub_fetchlog` and read through it to find out what the error was. If you can't decipher the output, ask an expert for help or email the nova-offline listserv and include excerpts from the logs that seem to indicate what went wrong. Be warned, some messages in the log files indicate errors that are, in fact, totally benign.
2. *Keep notes.* Keep detailed and organized notes about every problem you encounter and what the solution was. The notes may become obsolete pretty quickly, but this will save you the hassle of re-diagnosing the same issues over and over. This advice really applies much more generally than just to managing grid jobs.
3. *Be patient.* Sometimes things don't work very well one day and then get much better the next. The grid serves a lot of scientists, and we aren't always privy to everything that has to be done to maintain it. If there's an endemic problem, ask someone about it, but don't expect things to always run perfectly smoothly.

GENIE and GSimpleNtpFlux

GENIE is the neutrino interaction simulation software used by NOvA. Specifically, it is used to "generate" particles from neutrino interactions that are then passed to Geant4, which simulates their travel through the detector. NOvA-specific code that integrates with GENIE can be found in the external `EventGeneratorBase` package and in the `EventGenerator` package of the NOvASoft repo. The GENIE code documentation is also available here.

The use of GENIE in NOvA is well documented on this page.

In order to run, GENIE takes the incident neutrino flux as input. For simulations relevant to the UpMu search, the easiest input format to use is the GSimpleNtpFlux (the macro linked at the bottom of that page was used as the base for both runWimpSim and the atmospheric neutrino flux driver). This is a simple ROOT tree that contains (roughly) one entry per neutrino and some metadata describing the total flux included in the file. One important note is that while distances are generally provided in cm in NOvA, GENIE expects coordinates to be given in meters in GSimpleNtpFlux files.

An important step in simulating neutrino interactions is properly configuring the detector geometry, which is discussed on this page. In the context of the upward-going muon search, the current best estimate of a correct configuration is available in the EventGenerator/GENIE/prodgenie_wimp.fcl file. In general, this job configuration file should be used when producing simulated events from GENIE flux files created using either WimpSim or the atmospheric neutrino driver, which are discussed in part 4.

xrootd

Files that are stored on pnfs (as almost all grid output files *should* be) cannot be opened in root using the typical `root <path to .root file>`. Fortunately, if the path to the file is known or can be determined (see eg. the fetch_upmu.py script in part 3 below), a tool called xrootd can be used to open the file in root, provided the required permissions have been acquired (see the section on writing a setup_nova function, above)[6]. In particular, a script called `pnfs2xrootd` takes a file with full path as an argument, causes the file to be mounted in such a way as to be available to the root binary, and outputs a url that can be used to access the file. For example, if the file I want to view is at /pnfs/nova/scratch/users/ram2aq/DDUpMu_ntuples/big_file.root, I can do `root -l` `` `pnfs2xrootd /pnfs/<...same path as above...>big_file.root` ``[7] to open the file in Root.

Because of some trickiness with permissions, xrootd cannot be reliably used in a non-login shell like a screen session. This makes it difficult to write effective scripts that require access to many Root files on pnfs, as will be discussed in the following section on scripting.

The Runs Database

An experiment with high data throughput must carefully record running conditions for its data. Considerations like how much of the detector was in working condition, how long each run was, and how many beam spills (if performing a beam-based experiment) occurred during a particular run ultimately affect the analysis. In NOvA these and other data are recorded for each run in a PostgreSQL database. IF you have never used SQL or one of its many dialects before, this guide may be helpful.

Documentation for the NOvA database can be found here. Unfortunately, this page is outdated at the time of this writing. The only use-case for the DB so far has been to determine the livetime

[6] In fact, xrootd can be used for files on almost any file system accessible to the gpvm's, but pnfs is probably the only use-case necessary for this project.

[7] In a *sh shell, surrounding a command in backticks (` - to the left of the 1 on the keyboard) effectively replaces it with its output for the purposes of the surrounding command. So `` `pnfs2xrootd <path to file>` `` is replaced by the necessary url.

for a selection of subruns. This information is most efficiently accessed by querying the subruns table of the DB, but it can also be calculated by making a query to SAM for each file (see Zukai's UVa elog here). SAM is quite slow and sometimes unreliable, so this method is not preferred.

Unfortunately, the subruns table has only been consistently maintained since run 19301 (the first run used in UpMu analysis so far is 18399). The start and end times for all runs before 19301 were extracted, and sql commands prepared to insert this information into the DB, but the current status of this process is not known. For runs before 19301, you should use the information in /nova/ana/users/ram2aq/backfill_db/ and in this elog post. For runs after 19301, you can query the subruns table in the replicated DB (ifdbrep.fnal.gov) on port 5436. To open an interactive connection, use this command:

```
psql -h ifdbrep.fnal.gov -d nova_prod -p 5436 -U nova_reader
--password
```

You will then be prompted to enter the password for nova_reader, which you should obtain from the author (ram2aq@virginia.edu) or Jon Paley. This will start a psql session, and you will see this prompt:

```
nova_prod=>
```

The first thing you will need to do is to specify the search path to use:

```
set search_path to FarDet;
```

Remember that all SQL commands are terminated by a semicolon. If you forget the semicolon, you will see this prompt:

```
nova_prod->
```

Omitting the semicolon will thus allow you to break a single command over multiple lines. Once you have set the search path you can use standard SQL commands to access information from the database tables. Use `\dt` to list all tables and `\d <table name>` to get the columns in a particular table. The subruns table (`fardet.subruns`) has the start and stop times for all subruns after run 19301 (April 9, 2015), which is useful for determining the livetime for some sample of subruns. The tstart and tstop times (note that tstop is an optional column, so not every subrun will have a value) are in UTC.

As an example, to get the latest 10 subruns, you would do:

```
select run,subrun,tstart,tstop from subruns where run > 19301 order by
run DESC limit 10;
```

To exit the psql session, just do `\q`.

Part 3: Scripting

In general, computer programs can be written in one of two ways (although in reality there are many variations on each): as a script to be interpreted or as code to be compiled into an executable binary. An interpreter is a program that takes a script (sometimes alternatively called a macro, although this is technically incorrect) as input and executes each line of the script, one at a time and in order. Python and Bash are both examples of interpreted languages (although there exist compilers for Python code). C++ is not an interpreted language; to run a C++ program, it must first be compiled to create a binary.

ROOT allows users to create scripts/macros that are interpreted by CINT within the ROOT shell. Alternatively, users can choose to compile their scripts using ACliC. In general, compiled libraries created by ACliC will run faster than the same scripts when interpreted by CINT, especially if a high level of compiler optimization was used when compiling ROOT (as is true of the maxopt build on the NOvA gpvm's). This demonstrates the first principle of scripting:

- Scripts run slower than equivalent compiled binaries, especially for computationally expensive tasks.

As an example, consider the following ROOT script that reads 10,000 random integers from a file, sums them, and outputs the sum to another file (it repeats this 10,000 times to get a better estimate of running time):

```
#include <string>
#include <fstream>
#include <iostream>

const size_t ITERATIONS = 1e4;

void sum10krandoms(string infile, string outfile) {
  for (size_t iter=0; iter<ITERATIONS; ++iter) {
      std::ifstream inFile(infile.c_str(), std::ifstream::in);
      if (!inFile.good()) {
      std::cout << "Could not open file " << infile << std::endl;
      return;
      }
      std::ofstream outFile(outfile.c_str(), std::ofstream::out);

      unsigned long sum = 0;
```

```
      for (size_t i=0; i<10000; ++i) {
      unsigned x;
      inFile >> x;
      sum += x;
      } // i

      outFile << sum << std::endl;
      inFile.close();
      outFile.close();

      if (iter % 100 == 0) std::cout << 100.*iter/ITERATIONS << "%" <<
std::endl;
  } // iter
}
```

This script can be run from the Root prompt using CINT, or compiled into a shared library using ACliC. Running it with CINT on one of the gpvm's took 2m5.845s in the maxopt build (after starting root, just do `.L sum10krandoms.C` then `sum10krandoms("10krandoms.txt","out.txt")`). Running with a version compiled using ACliC took only 0m30.486s in the maxopt build (simply put a "+" at the end of the load command: `.L sum10krandoms.C+`). The times for the debug build were 3m58.478s for CINT and 0m33.703s for ACliC.

An equivalent Python script is:

```
import sys

for i in xrange(0,10000):
      infile = open(sys.argv[1], 'r')
      outfile = open(sys.argv[2], 'w')

      sum = 0
      for line in infile:
          sum += int(line.split()[0])

      outfile.write(str(sum))
      infile.close()
      outfile.close()
      if (i%100)==0:
            print str(i/100) + "%"
```

Running this script on one of the gpvm's took 3m19.147s in the maxopt build and 3m17.879s in the debug build (the two builds use the same python binary, which can be verified by doing `which python`). Neither of these scripts was written to be particularly efficient; often differences in time can be significantly reduced by coding more efficiently. The intention is that they should represent a typical level of skill for a physics student, and they solve a typical problem: how can I read many values from a file and process all of them somehow?

Note that the Python script, though slower, was only 12 lines long, while the Root macro was 21 lines. Note also that the Root macro was written so that it could be compiled, while if it were only going to be run with CINT, the `#include` lines and any use of "`std::`" could have been omitted. This exemplifies the second principle of scripting:

- Scripts are usually shorter and less verbose than equivalent compiled codes. They are thus often easier and quicker to write, debug, and maintain.

Evidently choosing whether to write a script or a compiled code involves balancing two considerations: How computationally expensive will this program be? And how much time do I want to spend making it? If you will be implementing[8] a sophisticated algorithm that you expect to have to run on many thousands of events, or it you need tools that are only available in an experimental software library (eg. the NOvA detector geometry code), you should probably write code that you will compile into a binary or library (like an art module). If you merely want to merge a few dozen, or even a few hundred, files in a predictable directory structure, consider if a short Bash or Python script would do the trick (hint: either will do).

Example UpMu Scripts, Binaries, and Libraries

Let's consider some specific cases from the UpMu project, examining the rationale behind the choice of language for each:

- The UpMuRecoAna module: A tool was needed to run on the reconstructed trigger sample and extract relevant data points for the tracks, hits, slices, and events. Since this program would take art files as its input, it was straightforward to choose an art module for the job. The module and the Root ntuples that it produces are described in Part 4.
- fetch_evd.py: a tool was desired to fetch reconstructed files from the sample stored in pnfs based on the SAM dataset tag, run number, and subrun number (for the datasets in question, only 1 file exists per run per subrun). The tool would then open the file for viewing in the event display. This requires a samweb query, some output parsing, then a second samweb query, and finally a way of interfacing the event display with the file in pnfs (either using xrootd or by copying it to a temporary location on bluearc). I chose to

[8] A bit of Computer Science jargon meaning "creating code to make use of"

implement this tool using Python since it was quick and required low overhead. Though executing system calls is slightly more onerous in Python than in (eg.) Bash, the output processing is more straightforward and familiar. Eventually I settled for printing the location of the file instead of opening it in the event display. In that way the script could be combined with other system calls from the shell by using backticks[9]. A copy of the script is at /nova/app/users/ram2aq/scripts/.

- As previously discussed in the section on SAM for Users, the add_dataset tool declares files to SAM with minimal metadata that is not sufficient to allow running art jobs on the grid. A tool was thus needed to generate SAM queries to update the metadata for each file. The necessary SAM query requires the metadata to be in json format. As there are many files (~53,000 in the through-going sample), many queries need to be executed. Each takes several seconds, meaning that whatever tool was developed would need to be run in a detached shell with screen or a similar utility. I chose to write a Python script for this task, as well, for similar reasons as for fetch_upmu: string output processing is straightforward and easy in Python, and system calls are only marginally more difficult than in Bash. The script can be found at /nova/app/users/ram2aq/scripts/update_mysam_metadata.py.
- Converting WimpSim output to NOvA flux files: WimpSim is a simulation program (actually, a suite of several programs), written in FORTRAN, that produces a simulated neutrino flux at a detector location from WIMP annihilations in the Sun and Earth. The flux is output as both differential fluxes in energy and angle bins and as individual neutrinos for an event-based monte carlo. A tool was needed to integrate the latter with existing NOvA simulation code, which required producing a flux file in a format that GENIE would recognize. WimpSim and the tools used to do this are described in detail in a section of part 4. Originally, several ROOT macros were developed to do this. They were written so as to be compiled with ACliC and run sequentially; three different scripts were needed to convert from WimpSim output to the GENIE flux format. Eventually, a single script was used to perform all three tasks, as well as making system calls to run the two WimpSim binaries. A current version of the script can be found at /nova/app/users/ram2aq/we2nova/runWimpSim.C. Nevertheless, the process was slow and difficult to automate, and thus unsuitable for generating a large simulated sample. Though tricky because of the need to link against both the FORTRAN WimpSim libraries and C++ GENIE libraries, a single standalone C++ binary was a much better solution, and one was eventually written. The details are discussed below in a section of part 4.

[9] In a *sh shell, surrounding a command in backticks (“`” - to the left of the 1 on the keyboard) effectively replaces it with its output for the purposes of the surrounding command. Ie. pnfs2xrootd `./fetch_evd.py <some dataset> <run number> <subrun> .` would print the url to use for opening a particular file in xrootd. Coincidentally, $(command) does the same thing, and is much easier to nest, eg. $(command1 $(command2)).

- Atmospheric neutrino simulation: In much the same way that the WimpSim to NOvA code was developed, several scripts were originally used to interface with atmospheric neutrino flux predictions. Unlike in the WimpSim case, no C++ binary has been created to improve on this process. Doing so would be very helpful for generating large signal samples in the future, as described in a later section in part 4.

Part 4: UpMu-specific tools

Atmospheric neutrino simulation

Atmospheric neutrinos are generated by interactions of cosmic rays in the atmosphere and the decays of the interaction products. Cosmic rays can be incident from any direction, and the neutrinos produced by their interactions are more likely than not to travel through the entire Earth without interacting. Effectively this means that atmospheric neutrinos can be incident on the NOvA detector from any direction at any time, implying that some of them should produce upward-going muons in the detector.

In the context of a WIMP annihilation search (see the next section on WimpSim), this represents a background process, but the atmospheric neutrinos are interesting in and of themselves because they allow oscillation studies that probe values of (L/E) that cannot be reached by beam oscillation experiments. Super Kamiokande was the first experiment to provide strong evidence of atmospheric neutrino oscillations. MINOS also performed a search for upward-going muons from atmospheric neutrinos and also saw evidence for oscillations in atmospheric neutrinos. Here are a few of the relevant papers:

- Super-K through-going muon oscillation paper (1999)
- Super-K stopping muon oscillation paper (1999)
- Super-K evidence for atmospheric neutrino oscillation (1998)
- MINOS atmospheric neutrino search (2012)
- A summary of atmospheric neutrino searches and techniques (2004)

In an event-based study, simulating atmospheric neutrino interactions at a detector involves two steps: first, the neutrino flux is estimated using a MC simulation of cosmic ray interactions in the atmosphere, and second, the flux is then used as the input to a neutrino interaction simulation. In theory it is possible to combine these steps, but in practice it is more convenient to split them up. Currently no neutrino flux estimations exist for Ash River, MN, but there are several that were prepared for Soudain, MN.

The existing scripts for producing simulated atmospheric neutrino interactions in NOvA are designed to work with the flux predictions of the Bartol group. These are the same predictions

used by MINOS in the previously linked paper (see references 30 and 31 in that paper). A description of the flux files, along with links to download them, can be found here. For neutrinos with energy below 10 GeV, a 3D simulation was used and the fluxes for both Soudain and Kamioka provided. For energies from 10 GeV to 10 TeV, a 1D simulation was used and only the flux for Kamioka provided. Additionally, because solar magnetic activity strongly influences the cosmic ray flux, predictions for solar maximum (highest activity -> smaller neutrino flux) and minimum are provided in separate sets of files. Finally, the energy-zenith angle distributions and energy-azimuthal angle distributions are provided in separate sets of files, although there are no azimuthal distribution predictions for the 1D simulation.

All existing simulated samples were produced using these files under the solar maximum assumption and combining the 3D simulation prediction at Soudain and the 1D prediction for high energies at Kamioka.

Starting from a fresh download of the Bartol flux files, these are the steps needed to produce NOvA events from simulated atmospheric neutrino interactions (note that a copy of each of these scripts can be found in `/nova/app/users/ram2aq/atmosnu`):

1. Concatenate the 4 ({nue, anue, numu, anumu}) high-energy flux prediction files to the appropriate low-energy prediction files. For example, to combine the solar maximum zenith-angle predictions (remember, no azimuthal distributions for high energy) for nue's, execute the following command:
   ```
   cat f210_3_z.kam_nue >> fmax20_i0403z.sou_nue
   ```
2. Produce the 8 ({nue, anue, numu, anumu} x {zenith, azimuth}) required flux tree (.root) files using the `load_bartol.C` macro. This will require 8 executions of the `load_bartol` function.
3. Combine the 8 trees into a single .root file. The trees must be named as indicated in the renameTrees.C script.
4. Execute the `bartolFluxDriver` function in the `bartolFluxDriver.C` macro. These are the arguments expected by the driver function:
   - `string outfile` - path to output file
   - `string infile` - path to .root file created in step 3
   - `bool fluxmode` - true to produce a GENIE-compatible flux file, false to produce a text file with information about each neutrino thrown
   - `long int nentries` - number of neutrinos to throw
   - `double enumin` - minimum energy of neutrinos produced
   - `bool doaux` - currently this has no effect and can be passed either true or false

   Alternatively, execute the `bartolFluxDriver_UpMu` function to produce only muon-flavor neutrinos with positive vertical component of momentum.

5. In a test release, modify the `EventGenerator/GENIE/prodgenie_wimp.fcl` file to use the GENIE flux file created in step 4. To do so, modify the `FluxSearchPaths` and the `FluxFiles` fields at the bottom. Build your test release using `make`.
6. Start an art job using the `GENIEGen` module to simulate the neutrino interactions: `nova -c job/prodgenie_wimp.fcl -n <desired number of evts> -o <desired output evt file>`

The flux driver interprets the number of neutrinos produced as an exposure time, and passes that time in μseconds into the flux ntuple in the `fmeta->protons` leaf. See the `init` and `init_UpMu` functions in the `bartolFluxDriver.C` macro for the details of this exposure estimation. When GENIE simulates the neutrino interactions, it internally reweights the interaction probabilities in order to produce a reasonable number of interactions. The effect of this reweighting is to increase the effective exposure of the sample (often by several orders of magnitude). Passing the exposure estimation into the protons field is necessary to obtain a reweighted exposure, which can then be used to predict interaction rates. GENIE reports the corrected number of protons (in this case interpreted as exposure in μseconds) at the end of the simulation art job, and it is included in the `SubRuns` tree of the produced art file, in the `sumdata::POTSum_generator_Genie.obj.totpot` leaf. This exposure estimation is also necessary for the WimpSim event simulation, where the exposure is interpreted as the number of annihilations.

The driver will also assign an MJD date and time to each neutrino, which is passed into the `fnumi->tpx` leaf. The fluxes are interpreted as uniform in time, so an MJD value is chosen from a uniform distribution between a min and max value set as parameters in the `bartolFluxDriver.C` macro. This value is of interest for the WIMP annihilation search since neutrinos from WIMP annihilations in the sun have angle zero relative to the sun whereas atmospheric neutrinos can have any angle relative to the sun. The date and time of an event are necessary to determine the location of the sun when the event occurred. Once GENIE has simulated the neutrino interaction, this value is available in the produced art file in the `simb::MCFlux` objects. An example of how to access the value can be found in `/nova/app/users/ram2aq/RunWimpSim_dev/Eval/SunDirAna_module.cc`.

There are several problems with the current implementation of the atmospheric neutrino flux driver:

- An atmospheric flux driver already exists as part of GENIE. Implementations exist for use with both the Bartol flux predictions and those using fluka. Furthermore, the NOvA GENIEHelper (and thus GENIEGen) is already configured to use these flux drivers, which require only properly formatted flux files to run. Initial tests of the flux distributions produced by these implementations showed that both agree with those given

in the relevant literature, indicating that on the GENIE side of things, the drivers work as expected. However, the calculation of the absolute normalization (or more cogently, the estimation of the exposure), which occurs in the GENIEHelper, is not straightforward and should be reworked and tested. Robert Hatcher has indicated a desire to do so.

- Steps 1 through 4 should be consolidated into a single C++ executable that should be built against current versions of GENIE and ROOT, in much the same way that runWimpSim is built (see the discussion in the next section). This will eliminate the need to edit one's .rootrc file and the need for a custom-compiled version of the GENIE library. The details of these requirements can be found in my elog post #40.
- Currently, in UpMu mode, the flux driver randomly places the neutrinos on a plane below the detector. The dimensions and location of this plane should be configured to best approximate the true angular distribution of incident atmospheric neutrinos. This issue is complicated by the fact that GENIE dynamically adjusts the size of the region in which it may generate interactions according to the energy of the flux neutrinos (as described in the a previous section on GENIE in part 2).

I recommend that a decision be reached regarding whether to continue to develop the NOvA-specific flux driver or to switch to the drivers provided by GENIE *before* significant work is put into the existing driver. This decision should be made in consultation with the simulations group and in particular with Robert Hatcher. If it is decided that the GENIE drivers should be used, then some effort will be needed to understand how the `GENIEHelper` is interfacing with them and how the exposure calculation should be done. On the other hand, if the decision is to further develop with the current implementation, then the first task should be to write a C++ program analogous to `runWimpSim` that could then be committed to the `EventGenerator` package. This program would need to link against GENIE and ROOT, but would be much simpler to write than `runWimpSim` since neither of these libraries are written in FORTRAN.

WimpSim

Dark matter comprises a large portion of the energy density of the observable universe, but its particle nature is not understood. Weakly-Interacting Massive Particles (WIMPs) are a popular candidate for dark matter. Should they exist, they would be non-relativistic (there are strict experimental and theoretical limits on the prevalence of relativistic dark matter particles). As astronomical bodies (stars, galaxies, planets, etc.) move through space, their gravity wells attract and trap WIMPs when they scatter on nuclear material within the bodies. There would then be a higher concentration of WIMPs within the center of large bodies compared to the surrounding space, and in the equilibrium condition the rate of annihilations of these WIMPs would equal the rate of capture.

These WIMP annihilations would produce high-energy photons and neutrinos that could, presumably, be detected by Earth-bound detectors. An excess of such high-energy particles coming from the center of massive astronomical bodies could thus be interpreted as evidence for the existence of WIMPs, and a detected lack of such an excess can inversely be interpreted as a limit on the WIMP annihilation cross-section. This is what is meant by an indirect dark matter search - NOvA and other experiments are looking for a high-energy neutrino (other times, photon) signal from dark matter particles in the Sun or other bodies.

Here are a couple helpful papers on this topic:

- SuperK indirect dark matter search results (2015)
- A discussion of WIMP theory and how experimental limits can be calculated (1996)

WimpSim is a FORTRAN library designed for use in indirect dark-matter searches. It simulates WIMP annihilations in the Sun and the Earth, as well as the propagation of neutrinos produced by these annihilations to a detector on Earth. To do this, it makes use of several libraries and has about a dozen dependencies that must each be compiled in order to produce a working installation. More details on how the simulation works can be found in this document, and the first few slides in this slidedeck summarize the inputs and outputs of the program's two binaries.

Since installing WimpSim on the vm's involves addressing several technical and somewhat complicated concerns, I will defer further discussion to Appendix B. The libraries are already installed in a ups product in the $EXTERNALS area on the vm's, and the runWimpSim.cpp program, which is part of the EventGenerator package, is a simple interface that produces the flux files that are needed as input to GENIE. If you intend to use your own installation of WimpSim to produce simulated events, refer to Appendix C. The remainder of this section will assume that the runWimpSim.cpp program is used[10].

There are essentially two steps needed to produce simulated NOvA events from WIMP annihilation-produced neutrinos:

1. Use `runWimpSim` to produce a GENIE-compatible neutrino flux file, as described in the previously-linked instructions.
2. (corresponding to steps 5 and 6 in the atmospheric neutrino section above) Modify the `prodgenie_wimp.fcl` file in the `EventGenerator` package to point to the flux file produced in step 1. Run a `nova` job using the modified configuration file.

[10] As of the time of this writing, runWimpSim has not been extensively tested. If you should encounter any problems while following the instructions posted here, please email the author at ram2aq@virginia.edu with details.

Though WimpSim can produce summary histograms with the expected neutrino flux as a function of energy, much like the atmospheric neutrino flux prediction, so far we have used the event-based flux prediction provided by `wimpevent` rather than the summary flux files. Ultimately this means a single GENIE flux file (as discussed in the preceding section on atmospheric neutrino simulation), containing some number of flux neutrinos, is interpreted as corresponding to a number of WIMP annihilations in the Sun (or in the Earth). This number is interpreted by GENIE as the total number of protons on target (pots) corresponding to the output of the simulation, and is available in the final simulated event file in the `SubRuns` tree and the `sumdata::POTSum_generator_Genie.obj.totpot` leaf.

To discriminate neutrinos from WIMP annihilations in the Sun from atmospheric neutrinos, it is important to know the location of the Sun relative to the detector. Therefore the date and time of the event must be known. The preceding section on atmospheric neutrinos details how this information can be extracted from the final simulated event file. Note that this process differs entirely from that used to extract the date and time of an event in actual data. For an example of how to extract date and time for both simulated and actual events, and of how to determine the Sun's location using the `NOVAS` library (available in the `MoonShadowTrigger` package of the DDT repository), see
`/nova/app/users/ram2aq/RunWimpSim_dev/Eval/SunDirAna_module.cc`.

UpMuAna and UpMuRecoAna

Let's say I have some collection of triggered events, and I want to know if there are *really* any upward-going muons in the sample. What I will probably have access to are artdaq format files containing the RawDigits[11] from all the cell hits in the triggered events, and I will need to process them somehow to try to find the upward-going tracks. I have essentially four choices to move forward:

1. Write my own tools to process the RawDigits and search for upward-going stuff. Starting from scratch like this is almost certainly not the best option.
2. Use the DDT framework modules that are used online by the trigger to produce hits, slices, and tracks and then write my own module that tries to reproduce what the trigger does. This module already exists in the Eval package and is called UpMuAna.
3. Use the offline analysis framework to run "reconstruction" on the artdaq file to produce a reco file with various objects like CellHits (calibrated hit objects), Clusters (slices), and tracks of several varieties. Then write my own module that examines these objects and

[11] There will also be a std::vector<rawdata::RawTrigger> for each event which will contain the time window (in TDC) that the UpMu trigger decided contained the upward-going track. This information will certainly be useful for verifying the efficiency of the trigger, but so far has not been used. There is also a rawdata::DAQHeader which contains information about the run configuration and time of the event.

looks for upward-going muons however I may decide to do that. This module also exists in the Eval package and is called UpMuRecoAna.

4. Run "reconstruction" like in option 3, but instead of using the reconstructed objects, continue to run the offline analysis tools like PID and CAFAna to ultimately produce a CAF files, which I can then write ROOT scripts to examine. Unless I modify the modules myself, I do not have direct control of what information about the tracks is stored in the final output files, and it is not possible to store hit-level information.

Refer to the previous sections on SAM, samweb, and xrootd to learn how to locate and acquire/run on triggered files.

So far in the course of this project, we have used options 2 and 3. Before reading on, if you are unfamiliar with the techniques (hit time estimates, slope, LLR, etc.) used to identify upward-going muons, please review this note and these proceedings from DPF 2015. If you are curious about naming and numbering conventions for detector geometry-related objects (Diblock, DCM, plane, cell, etc.), see this note.

UpMuAna instructions

To pursue option 2, you must have a partial or complete build of the DDT repository on one of the vm's. There are two ways to do this, and the currently preferred method is to use cmake. However, at the time of this writing I do not know how to source a cmake installation for use on the grid, so I will be detailing the method that uses only srt. Here are some instructions for setting up and running the UpMuAna module:

```
setup_nova
newrel -t development DDTAna_dev #-> this sets up a new test release
in your nova/app/ directory
cd DDTAna_dev

# we will need a bunch of packages from the Data-Driven Trigger
software repository
# these lines "check out" those packages into your local release so
you can compile them
# you can explore the separate DDT repo here
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h HitSorters
```

```
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h Tracking
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h Statistics
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h CalibrationTriggers
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h DDTBaseDataProducts
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h DDTGlobalConfigurations
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h HitSlicers
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h NuMuTriggers
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h PatRec
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h DAQUnpackUtils
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h DDTUtilities
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h DDTCore
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h ExoticsTriggers
addpkg_svn -d
svn+ssh://p-novaddt@cdcvs.fnal.gov/cvs/projects/novaddt/novaddt.svn/
-h SRT_NOVADDT
```

```
# the following lines "check out" the latest version of some packages
that we will need from the NOvASoft repo.
# You can explore the repo here.
addpkg_svn -h SRT_NOVA
addpkg_svn -h DDTConverters
addpkg_svn -h Eval
addpkg_svn -h Utilities

# finally, we need some packages from the Data Acquisition repo
addpkg -hd :pserver:anonymous@cdcvs.fnal.gov:/cvs/nova DAQDataFormats
addpkg -hd :pserver:anonymous@cdcvs.fnal.gov:/cvs/nova DAQChannelMap
addpkg -hd :pserver:anonymous@cdcvs.fnal.gov:/cvs/nova
NovaDAQConventions
addpkg -hd :pserver:anonymous@cdcvs.fnal.gov:/cvs/nova PackageVersion
```

Now we need to make a small change because the UpMuAna module (the one we want to run) relies on an ART service from the DDT repo. You'll need to edit the Utilities/services.fcl file. In the includes section, add:

```
#include "WaveformProcessor.fcl"
```

In the block that says "standard_services:", add:

```
WaveformProcessor:  @local::standard_waveformprocessor
```

Now in the terminal, do:

```
srt_setup -a # ->this tells srt, the compiler system we use in NOvA,
to look in your local test release first.
# You'll need to do this anytime you change code locally and want to
compile

make # -> this compiles the code in all the packages
# it took me ~20 minutes to compile everything

# now we can run the module on some far detector data:
```

```
nova -c job/upmuanajob.fcl -s
/nova/ana/trigger/data/fd/S14-09-29/141110_Streamer_Run_18115/fardet_r
00018115_s00_t04.root -n 1
# the -c flag is used to specify the configuration file (a FHICL file,
which you can read about here)
# the -s flag specifies which data file to run on
# here I have chosen to run on only 1 event by using the -n flag (you
should use more)
```

This should produce a file called UpMuAna_Hist.root, which will contain a TDirectory called `upmuana` with three ntuples in it. Here are the details about what each ntuple will contain:
`particle_ntuple` (if the `IsSim` fcl parameter is set to true, which should only be done for simulated events)

Branch name	*Description*
Run	The run number
Event	The event number (unique per run)
ParticleID	A unique integer identifying the particle in the event
PDGid	The PDG code of the particle
InitKE	Initial kinetic energy of the particle
endKE	The kinetic energy of the particle at it's end
MuEneDep	The total energy (GeV) deposited by this particle, if it is a muon, else 0.
MuInitKE	The initial kinetic energy (GeV) of this particle if it is a muon, else 0.
Mother	The unique ID of the particle's mother.
Process	0 if a primary muon or neutrino 1 if a mu- decay 2 if a mu+ decay

`hit_ntuple`

Branch name	*Description*

Run	The run number.
SubRun	The subrun number.
Event	The event number (unique per run).
TrackID	A unique (per event) integer identifying the track to which this hit belongs.
iHit	Before fitting a line to the measured times versus expected times for the track, each hit is sorted according to its height in the detector (lower hits first). This is the number of the hit within this sorted order (0-indexed). A hit is uniquely identified by its run, event, track, and hit numbers.
View	The view (0->XZ or 1->YZ).
eT	The expected time for the hit (ns since the trigger time, negative values allowed).
mT	The measured time for the hit with all of our calibration code applied (ns since the trigger time, negative values allowed).
mT_uncor	The measured time for the hit with no calibration applied (ns since the trigger time, negative values allowed).
compMT	The calibrated measured time, converted into a basis for comparison with the truth time (truthT).
truthT	The true time of the hit (for simulated events only). This is the arithmetic average of the sim::FLSHit::GetEntryT and sim::FLSHit:GetExitT, assuming there was only one hit in the channel (otherwise it is ambiguous to which of the multiple hits the FLSHit objects correspond). If there were multiple hits, then the value is 0 (do not use it for comparison).
PDG	The PDG code associated with the sim::FLSHit identified with this hit. If no FLSHit was identified, the value is -999.
TDC	In TDC, the time of this hit minus the time of the first hit in the track (hit with lowest y-coordinate).
ADC	The ADC of the hit. Note that hits with ADC<50 are skipped.
Plane	The plane number of the hit.
Cell	The cell number of the hit.

DCM	The number of the DCM that contains the hit.
X	The approximate X-coordinate in cm in the NOvA coordinate system.
Y	The approximate Y-coordinate in cm in the NOvA coordinate system.
Z	The approximate Z-coordinate in cm in the NOvA coordinate system.
D	The approximate distance of the hit along the cell direction (vertical for XZ-view and horizontal for YZ-view) from the read-out electronics (used to calibrate both hit time and signal attenuation). This is estimated by assuming a linear track and performing a linear extrapolation using the z-coordinate of the hit.
TrackLen	The length of the track to which this hit belongs, calculated by computing the distance between the first and last points under the y-coordinate sorting.

`track_ntuple`

Branch name	*Description*
Run	The run number.
SubRun	The subrun number.
Event	The event number (unique per run).
StartX	The starting x-coordinate (in the NOvA coordinate system, cm) of the track.
StartY	The starting y-coordinate (in the NOvA coordinate system, cm) of the track.
StartZ	The starting z-coordinate (in the NOvA coordinate system, cm) of the track.
EndX	The stopping x-coordinate (in the NOvA coordinate system, cm) of the track.
EndY	The stopping y-coordinate (in the NOvA coordinate system, cm) of the track.

EndZ	The stopping z-coordinate (in the NOvA coordinate system, cm) of the track.
dTX	The time difference in TDC between the first and last hits (lowest and highest y-coordinates, respectively) in the XZ-view.
dTY	The time difference in TDC between the first and last hits (lowest and highest y-coordinates, respectively in the YZ-view.
TrackLen	The length of the track in cm, defined as the magnitude of the vector between the first and last points.
TrueLenght	*(note the incorrect spelling of length)* The length of the track in the detector, taken from the truth information (else -999).
TrueEDep	The total energy deposited in GeV, taken from the truth information (else -999).
TrackID	A unique (per event) integer identifying the track to which this hit belongs.
R2X	The R^2 value of a linear fit to the XZ-view cell numbers vs. plane numbers. A value close to 1 indicates a very linear track, a value less than 1 indicates a more curved track.
R2Y	The R^2 value of a linear fit to the YZ-view cell numbers vs. plane numbers. A value close to 1 indicates a very linear track, a value less than 1 indicates a more curved track.
nXhits	Number of hits on the track in the XZ-view.
nYhits	Number of hits on the track in the YZ-view.
SlopeX	The best-fit slope of a linear fit to the measured times vs expected times for hits in the XZ-view only.
SlopeY	The best fit slope of a linear fit to the measured times vs expected times for hits in the YZ-view only.
SlopeXY	The best fit slope of a linear fit to the measured times vs expected times for all hits.
Chi2X	The reduced χ^2 value for the linear fit to the measured times vs expected times for hits in the XZ-view only.
Chi2Y	The reduced χ^2 value for the linear fit to the measured times vs expected times for hits in the YZ-view only.

Chi2XY	The reduced χ^2 value for the linear fit to the measured times vs expected times for all hits.
ProbUpX	The probability of the track being upward-going from the linear fit with fixed slope +1 to the measured times vs expected times for hits in the XZ-view only.
ProbDnX	The probability of the track being downward-going from the linear fit with fixed slope -1 to the measured times vs expected times for hits in the XZ-view only.
ProbUpY	The probability of the track being upward-going from the linear fit with fixed slope +1 to the measured times vs expected times for hits in the YZ-view only.
ProbDnY	The probability of the track being downward-going from the linear fit with fixed slope -1 to the measured times vs expected times for hits in the YZ-view only.
ProbUpXY	The probability of the track being upward-going from the linear fit with fixed slope +1 to the measured times vs expected times for all hits.
ProbDnXY	The probability of the track being downward-going from the linear fit with fixed slope -1 to the measured times vs expected times for all hits.
ProbUpRoot	The probability of the track being upward-going from the linear fit with fixed slope +1 to the measured times vs expected times for all hits, using the Root TMath library.
ProbDnRoot	The probability of the track being downward-going from the linear fit with fixed slope -1 to the measured times vs expected times for all hits, using the Root TMath library.

The UpMuAna module can be run on the grid using the instructions in the previous section on the NOvA grid submission scripts, provided that you have first set up a working installation in a test release and that you specify the release in the job configuration file.

UpMuRecoAna instructions

The UpMuRecoAna module functions in much the same way as the UpMuAna module, except that it uses data objects provided by the reconstruction modules. These modules automatically

calibrate hit time and energy, meaning that there is no need to correct for DCM time offsets or light propagation effects within the analysis code itself. They also contain more sophisticated tracking algorithms (and a number of varieties, although so far only one has been explored) than are used in the trigger code. To see (or edit) the data objects needed by the module, examine the Eval/UpMuRecoAna.fcl file and the module itself: Eval/UpMuRecoAna_module.cc.

You can install the module by simply creating a new test release, adding the Eval package, and doing `make`. You can then run it by doing `nova -c job/upmuanajob_reco.fcl /nova/ana/users/ram2aq/DDUpMu_reco/fardet_r00018976_s24_ddupmu_FA15-02-09_v1_data.reco.root`. The data file in that command is one subrun with 151 events that passed the DDUpMu (through-going) trigger, and it should take about 5 minutes to run. Once finished it will produce a file called UpMuRecoAna_Hits.root that will contain a TDirectory (upmureocoana) with 5 ntuples: track_ntuple, hit_ntuple, vertex_ntuple, event_ntuple, and slice_ntuple. Here is a description of their contents (I will omit the vertex ntuple as it has not been used and does not appear to be populating correctly):

event_ntuple

Run	The run number.
SubRun	The subrun number.
Event	The event number (unique per run).
Ntracks	Number of reconstructed tracks in the event.
NcontainedT	Number of fully-contained tracks in the event.
Nvertices	Number of reconstructed vertices in the event.
NcontainedV	Number of vertices within the detector.
avgNhits	Average number of hits per track in the event.
avgNRecohits	Average number of hits per track that could be properly calibrated.
avgLen	Average length for tracks in the event (cm).
hasUpMu	Does the event contain a track passing all cuts defined in the fcl file? (1 for true, 0 for false)
hasFCUpMu	Does the event contain a track passing all cuts defined in the fcl file that is also fully contained? (1 for true, 0 for false)

hasTGUpMu	Does the event contain a track passing all cuts defined in the fcl file that is also through-going? (1 for true, 0 for false)
nDiblocks	Number of active diblocks in the event (an active diblock is one with at least one hit within it).

slice_ntuple

Run	The run number.
SubRun	The subrun number.
Event	The event number (unique per run).
SliceID	The slice number (unique per event).
Ntracks	Number of tracks in the slice.
containment	4 if every track in the slice is fully contained, else 1.

track_ntuple

Run	The run number.
SubRun	The subrun number.
Event	The event number (unique per run).
SliceID	The slice number (unique per event).
TrackID	The track number (unique per event).
Nhits	The total number of CellHits in the track.
NRecohits	The number of CellHits that were succesfully calibrated as RecoHits.
NOutliers	The number of RecoHits that were treated as outliers for the measured time vs expected time linear fit.
ProbUp	The probability that this track is an upward-going muon from the linear fit with fixed slope +1.
ProbDn	The probability that this track is a downward-going muon from the linear fit with fixed slope -1.

LLR	The natural log of ProbUp/ProbDn (“Log Likelihood Ratio”).
Chi2	The reduced χ^2 value for the linear fit to the measured times vs expected times for all RecoHits.
Slope	The best fit slope of the linear fit to the measured times vs expected times for all RecoHits.
LLRX	The natural log of ProbUp/ProbDn (“Log Likelihood Ratio”) for hits in the XZ-view only.
Chi2X	The reduced χ^2 value for the linear fit to the measured times vs expected times for hits in the XZ-view only.
SlopeX	The best fit slope of the linear fit to the measured times vs expected times for hits in the XZ-view only.
LLRY	The natural log of ProbUp/ProbDn (“Log Likelihood Ratio”) for hits in the YZ-view only.
Chi2Y	The reduced χ^2 value for the linear fit to the measured times vs expected times for hits in the YZ-view only.
SlopeY	The best fit slope of the linear fit to the measured times vs expected times for hits in the YZ-view only.
R2X	The R^2 value of a linear fit to the XZ-view cell numbers vs. plane numbers. A value close to 1 indicates a very linear track, a value less than 1 indicates a more curved track.
R2Y	The R^2 value of a linear fit to the YZ-view cell numbers vs. plane numbers. A value close to 1 indicates a very linear track, a value less than 1 indicates a more curved track.
StartX	The x-coordinate (in NOvA coordinate system, cm) for the RecoHit with lowest y-coordinate in the track.
StartY	The y-coordinate (in NOvA coordinate system, cm) for the RecoHit with lowest y-coordinate in the track.
StartZ	The z-coordinate (in NOvA coordinate system, cm) for the RecoHit with lowest y-coordinate in the track.
StartT	The time in ns for the RecoHit with lowest y-coordinate in the track.
EndX	The x-coordinate (in NOvA coordinate system, cm) for the RecoHit with highest y-coordinate in the track.

EndY	The y-coordinate (in NOvA coordinate system, cm) for the RecoHit with highest y-coordinate in the track.
EndZ	The z-coordinate (in NOvA coordinate system, cm) for the RecoHit with highest y-coordinate in the track.
EndT	The time in ns for the RecoHit with highest y-coordinate in the track.
TrackHitsX	Number of RecoHits in the XZ-view.
TrackHitsY	Number of RecoHits in the YZ-view.
Length	Distance from RecoHits with lowest and highest y-coordinates in the track (cm).
dirX	The average of the interpolated track direction vector x-component for every RecoHit on the track (used in calculating elevation angle).
dirY	The average of the interpolated track direction vector y-component for every RecoHit on the track (used in calculating elevation angle).
dirZ	The average of the interpolated track direction vector z-component for every RecoHit on the track (used in calculating elevation angle).
eleAngle	The elevation angle of the track, computed from the average track direction vector for every RecoHit on the track. Use the absolute value of this number when cutting on elevation angle.
totalE	Total reconstructed energy deposition for all RecoHits on the track in GeV.
containment	4 if the track is fully-contained, 3 if it is in-produced (hit with lowest y-coordinate is inside detector, hit with highest y-coordinate is near the edge), 2 if stopping (entered detector from outside), else 1 -> fully contained.
avgTX	Average time of all RecoHits in the XZ-view in ns.
avgTY	Average time of all RecoHits in the YZ-view in ns.
meRecoGeV	The energy of the reconstructed Michel electron for this track if one exists, else -5.
meRecoNHits	The number of CellHits associated with a reconstructed Michel electron for this track if one exists, else -5.
meRecoX	The x-coordinate of the reconstructed Michel electron for this track if one exists, else -5.

meRecoY	The y-coordinate of the reconstructed Michel electron for this track if one exists, else -5.
meRecoZ	The z-coordinate of the reconstructed Michel electron for this track if one exists, else -5.
meRecoDist	The distance of the reconstructed Michel electron from the end of this track if one exists, else -5.
meRecoDeltaT	The time difference in ns between the end of the muon track and the reconstructed Michel electron if one exists, else -5.
meRecoAtTrackEnd	If the reconstructed Michel electron is at the end of the reconstructed muon track, then 1, or if not at the end 0, or if none exists, -1.

hit_ntuple

Run	The run number.
SubRun	The subrun number.
Event	The event number (unique per run).
TrackID	The track number (unique per event).
HitID	The hit number (unique per track).
X	The x-coordinate (in NOvA coordinate system, cm) of this hit.
Y	The y-coordinate (in NOvA coordinate system, cm) of this hit.
Z	The z-coordinate (in NOvA coordinate system, cm) of this hit.
T	The time (in ns) of this hit, with full timing calibration (*ie*. taken from the RecoHit object if it was correctly calibrated, else the CellHit object).
deltaT	The uncertainty on the time (in ns) of this hit, from the Calibrator getRes function using the calibrated PE.
eT	The expected time of this hit under the upward-going muon assumption, used in the mT vs eT fits.
PE	The energy of the hit (somewhat analogous to ADC).
PECorr	A “corrected” energy for the hit. WARNING: as of the time of this writing (March 2016), PECorr may be deprecated very soon. It has not yet been

	used in the analysis.
ADC	The uncalibrated energy of the hit obtained from the DAQ (used in DDT but not in offline analysis).
Plane	The plane number of the hit.
Cell	The cell number of the hit.
View	The geo:View_t number of the hit.
GoodTiming	Whether the calibrated time of the RecoHit was "good" or not, taken from the χ^2 of the fine timing fit of the theoretical trace to the APD readouts. 1 if true, else 0. Hits without "good" timing are not used in the mT vs eT fits.
Reco	Whether this hit was successfully calibrated, 1 if true, else 0.
Outlier	Whether this hit was considered an outlier or not for the mT vs eT fit, 1 if true, else 0.
dirX	Analogous to the x-component of the particle's momentum at the location of this hit, but in arbitrary units.
dirY	Analogous to the y-component of the particle's momentum at the location of this hit, but in arbitrary units.
dirZ	Analogous to the z-component of the particle's momentum at the location of this hit, but in arbitrary units.
eleAngle	The elevation angle of the track at the location of this hit. Calculated using atan(dirY/sqrt(dirX^2+dirZ^2)).
GeV	The calibrated energy of this hit in GeV, taken from the RecoHit. If the hit was not successfully calibrated, then 0.
Diblock	The diblock number of this hit.

If the input file was a simulated event and the IsSim fcl parameter is set to true, then a sixth ntuple will be present that contains some relevant truth information:

particle_ntuple

Run	The run number.

SubRun	The subrun number.
Event	The event number (unique per run).
ParticleID	A unique integer ID for this particle (unique per event).
PDG	The PDG code of this particle.
Length	The length of the particle (*not* within the detector), taken from the StartX, EndX, etc, of the sim::Particle object.
StartX	The x-coordinate of the starting point of this particle (in NOvA coordinate system, cm). Not necessarily within the detector bounds.
StartY	The y-coordinate of the starting point of this particle (in NOvA coordinate system, cm). Not necessarily within the detector bounds.
StartZ	The z-coordinate of the starting point of this particle (in NOvA coordinate system, cm). Not necessarily within the detector bounds.
StartT	The time when this particle started, in ns.
StartE	The energy of this particle when it started, in GeV.
EndX	The x-coordinate of the stopping point of this particle (in NOvA coordinate system, cm). Not necessarily within the detector bounds.
EndY	The y-coordinate of the stopping point of this particle (in NOvA coordinate system, cm). Not necessarily within the detector bounds.
EndZ	The z-coordinate of the stopping point of this particle (in NOvA coordinate system, cm). Not necessarily within the detector bounds.
EndT	The time when this particle stopped, in ns.
EndE	The energy of this particle when it stopped, in GeV.
StartPX	The x-component of the momentum of this particle in GeV/c.
StartPY	The y-component of the momentum of this particle in GeV/c.
StartPZ	The z-component of the momentum of this particle in GeV/c.
StartEleAngle	The elevation angle of this particle at its starting point.
containment	4 if the particle is fully-contained, 3 if it is in-produced, 2 if stopping (entered detector from outside), else 1 -> fully contained.

MotherID	ParticleID code for the mother particle, if it exists (value unknown if no mother exists).
TrackID	The ID code of the track produced by this particle.
EDep	The total energy deposited by this particle in the detector in GeV.
nFLS	The number of fls hits associated with this particle.

For an example of a script that accesses information from some of these ntuples, see /nova/app/users/ram2aq/scripts/ddcontained/find_contained_candidates.C.

Part 5: UpMu Datasets

This final section will document the datasets used for the most recent UpMu analysis. For information on defining datasets and using them in grid jobs, see the sections on SAM, SAM for Users, and NOvA grid submission scripts in Part 2.

You can obtain a list of all 40 datasets (in chronological order by the date they were defined) used so far in this project with this command: `samweb list-definitions --user ram2aq`

Most of the datasets were not used for the most recent analysis results or are simply recovery used to run recovery jobs, so they will not be included here.

You may notice that the artdaq dataset names listed here contain "S14-08-19". This is because for each raw file (of which there is exactly one per subrun), there are multiple artdaq files corresponding to different versions of the DAQ2RawDigit module. Though it is possible to run reconstruction on any one of these files, two reconstructed event files with different DAQ2RawDigit versions cannot subsequently be merged. It is therefore advisable to require a particular version of this module when defining a new dataset using the "`daq2rawdigit.base_release`" constraint.

Name	**File type**	**Number of files**	**Location**	**Approx. size**	**Description**
ram2aq_fd_artdaq_ddupmu_gt18398_lt19652_goodruns_S14-08-19	artdaq	53,840	Tape (not user-produced)	0.3 TB	All DDUpMu artdaq subrun files used in the analysis so far, with run range given in the name.
ram2aq_fd_artdaq_ddcontained_gt18398_lt19652_goodruns_S14-08-19	artdaq	53,793	Tape	3 TB	All DDContained artdaq subrun files used in the analysis so far, with run range given in the name.
ram2aq_fd_artdaq_ddcontained_gt18398_lt18819_goodrun	artdaq	19,033	Tape	1 TB	Approximately the first third of the DDContained

s_S14-08-19					artdaq subrun files.
ram2aq_fd_artdaq_ddcontained_gt18818_lt19239_goodruns_S14-08-19	artdaq	15,949	Tape	0.9 TB	Approximately the second third of the DDContained artdaq subrun files.
ram2aq_fd_artdaq_ddcontained_gt19238_lt19652_goodruns_S14-08-19	artdaq	18,811	Tape	1 TB	Approximately the last third of the DDContained artdaq subrun files.
ram2aq_reco_ddupmu_r18398-r19652	reco	55,944	Tape /pnfs/nova/users/ram2aq/UpMuRecoAna_r18398-r19652/	0.9 TB	All of the reconstructed DDUpMu subrun files, but with mixed DAQ2RawDigit versions, so they cannot be merged.
ram2aq_reco_ddcontained_r18398-r18819	reco	351	Tape /pnfs/nova/users/ram2aq/ContainedReco_r18398-r18819/	4.4 TB	The first third of the DDContained reco files, merged by run.
ram2aq_reco_ddcontained_r18818-r19239	reco	274	Tape /pnfs/nova/users/ram2aq/ContainedReco_r18818-r19239/	3.7 TB	The second third of the DDContained reco files, merged by run.
ram2aq_reco_ddcontained_r19238-r19652	reco	340	Persistent dCache /pnfs/nova/persistent/users/ram2aq/ContainedReco_r19238-r19652	4.9 TB	The last third of the DDContained reco files, merged by run.

Glossary

- ADC: "Analog-to-Digital Conversion" of the height of the signal waveform in the cell readout electronics. This is an unsigned integer that ranges from 0 to 4095, and is used as a measure of the amount of energy deposited within the cell. In the DDT, this unit is used, while in the offline analyses, the PE (which is extrapolated from the ADC) is the preferred unit.
- APD: An "Avalanche Photo-Diode" is a device that converts light incident on one end into an electronic signal. In NOvA, APDs are used to convert scintillation light from the liquid scintillator into a signal that can be digitized and stored for later analysis.
- Bash: Bourne Again SHell. A shell is an interface that allows a user to interact with the OS (see Wikipedia's article on shell). Bash is the GNU shell.
- Cell: see *channel*. Also, a variable that defines the horizontal or vertical position of a hit, if the hit is in the XZ or YZ view, respectively.
- CellHit: see also Hit, RawDigit. A calibrated version of a RawDigit (inheriting directly from RawDigit) that has the time in ns in the event time scale, after applying the DCM offset correction. It also has an estimate of the PE for the hit based on the ADC.
- Channel: one hollow PVC tube, measuring (approx.) 3.5cm x 6.654cm x 15.4m in the far detector, which is filled with pseudocumene-doped mineral oil scintillator and represents the smallest detector element. A wavelength-shifting optical fiber twice the length of the PVC is looped within the tube and both ends are attached to the readout electronics at one end of the tube. Each channel is independently sampled by an APD twice per microsecond (in the far detector) to measure the light level within the fiber.
- Cluster: see Slice.
- DAQ: "Data AcQuisition" is a blanket term describing the electronics and computing systems that convert physical activity within the detector into digital information that can be processed and stored for later use. The technical details of the DAQ are beyond the scope of this document, but you can find information about the data products produced by the DAQ in docdb#4390.
- DAQHit: see also Hit. The DAQHit is the DDT's object storing information about a single cell hit in an event. The existence of a DAQHit indicates that the DCS algorithm registered activity in the cell consistent with a particle depositing energy there. DAQHits store the plane, cell, view, ADC, and TDC of the hit. Here is the header file defining the DAQHit interface.

- DCS: The “Detector Control System” is an interface for controlling power distribution to different parts of the detectors, the cooling system needed to keep APDs at operating temperature, and for monitoring the temperature and power status of the detectors.
- DCS: “Dual-Correlated Sampling” is the algorithm used to determine if activity within a cell represents a physical particle “hit.” The memory footprint of events is dominated by hit-level information, so recording information about every channel in the detector for each readout is unfeasible. DCS compares the magnitude of the electronics response at each readout with the magnitude from 1 usec before. If the difference is more than some threshold, a hit is triggered and the information for the 4 most recent readouts is stored by the DAQ.
- DDT: The software (level 3) “Data-Driven Trigger” performs basic and highly optimized reconstruction and analysis on the live data from the detector in real time to select interesting events that should be stored permanently. See the more extensive description in Part 1.
- FHiCL: “Fermilab Hierarchical Configuration Language.” Part of the art software framework that defines syntax to use in writing job and module configuration files, which have the .fcl extension. The Mu2e collaboration has an excellent writeup describing the language and how it is used here. Explore the NOvASoft repository for many examples of module and job configuration files.
- Grid (The): a distributed computing system that allows users to submit jobs which will then be executed by one (or more) of multiple (at Fermilab, thousands of) independent “nodes.” Software (at Fermilab, jobsub, SAM) manages the allotment of nodes and the copying of input and output.
- Hit: A deposition of energy into a cell in the detector by a particle, or one of the code objects describing this deposition.
- jobsub: The tool used by the experiments at Fermilab to manage grid job submissions. See this page.
- node: See Grid. A CPU/RAM unit that can execute a job.
- PDG code: Particle Data Group code identifying the type of a particle (lepton, quark, meson, hadron, etc.). See this document for details.
- PE: The number of “Photo-Electrons” measured by the detector from a charged particle traveling through the scintillator in a single channel. This is used to determine the energy deposited by the particle in the cell.
- PECorr: The “Corrected” number of “Photo-Electrons” created as a charged particle travels through a single channel in the detector, taking into account the attenuation of the light signal as it travels along the fiber from the point where the particle was incident on the channel to the readout. This is a reconstructed quantity that depends on merging the information for a physics event between the two views.

- Plane: Cells in NOvA are arranged into flat planes 15.4m wide that are then stacked in alternating horizontal and vertical orientations along the beam (z-) axis. The plane number thus gives the z coordinate for a hit.
- pnfs: parallel network file system. pnfs doesn't work like the more typical file system on your laptop, and the details are well beyond the scope of this document. You should see this tutorial for guidelines on using it.
- RawDigit: A low-level representation of a Hit. It contains equivalent information to a DAQHit, but in slightly different format.
- RecoHit: see CellHit. A fully calibrated representation of a Hit that contains the calibration present in the CellHit, with an additional calibration in both time and energy based on an estimation of the distance of the hit from the channel readout. As the scintillation light propagates along the fiber in the channel, the light attenuates and takes some time. Both of these effects are taken into account when constructing a RecoHit from a CellHit.
- Repository: Or "repo." A software management system that stores source code and makes it available to developers for editing, generally as part of a version control system. Developing code for this project may mean interfacing with two repositories: the NOvADDT repo and the NOvASoft repo.
- SAM: The data access management system for the experiments at Fermilab. See the links on this page for more.
- samweb: A command-line interface to SAM.
- Slice: A "Slice" (called a "Cluster" offline) includes all hits that are close to one another in both time and space. Slice parameters are optimized to put all hits from one physical event (eg. a muon track, a neutrino interaction and subsequent shower, etc) together while including as few noise hits as possible.
- SQL: Structured Query Language. A computer language for interfacing with databases, allowing storage, manipulation, and retrieval of data.
- TDC: A "Time to Digital Converter" is a 'clock' that increments a counter by 1 at a set frequency. In NOvA the clock ticks at 64 MHz, setting the conversion factor 1 TDC = 15.625 ns. Evan Niner's technical note, docdb#12570, on the timing calibration contains details on the NOvA timing system.
- Track: In the context of this search, a "Track" is a reconstructed object corresponding to a relativistic muon traveling through the detector. Such muons generally travel in straight lines, so the defining characteristic of a muon event is a number of hits along a long, straight line. The DDT has its own Track object, while the Track objects used in the UpMu analysis are those defined in the NOvASoft repository, here.
- UpMu: upward-going muon.
- View: The horizontal planes in the NOvA detector define the "YZ View," which gives the vertical position of a hit and it's position along the beam axis.

Equivalently, the vertical planes define the “XZ View.” These two views can be combined to provide 3D tracking for particles in the detector.

- WIMP: Weakly-Interacting Massive Particle. A leading theoretical candidate for dark matter. If they exist, they should be annihilating at some rate that is higher in the center of the Sun than in the surrounding space. We want to see a high-energy neutrino signal from that annihilation. See the section on WimpSim and the Wikipedia article for more.
- XrootD: Software that provides access to files stored on different file systems. See the explanation in Part 2.

Appendix A: Introduction to Linux and the terminal

If you are a typical Mac or Windows user, you've probably never interacted with a terminal (command line interface) or with a shell like Bash. You will find that, in order to proceed with this or almost any other physics research project, you will need to be familiar with *NIX environments (like Linux, UNIX, etc.). Certainly, any substantial effort on NOvA will require that you be able to log into and interact with the gpvm's (described in Part 1), which are running a specific distribution (think "flavor") of Linux called SLF ("Scientific Linux Fermi").

If you are in need of an introduction or a refresher in using the terminal, there is an excellent tutorial available here that introduces all the basic concepts you will need to get started. You may wish to bookmark the page, as many of the commands (eg. sed, cut, grep) are incredibly useful but somewhat tricky to master. Section 6, on `vi`, is unnecessary if you already use a text editor like `emacs`. If you have never used a text editor, `gedit` (or `nedit`, if the former is unavailable) provides a simple and straightforward interface that is probably the easiest to master of those available on the NOvA vm's. Probably, you will not want to use an IDE for your work on NOvA; in text editing over an ssh connection, simpler is usually better.

The previously-linked tutorial briefly discusses how to obtain Linux. For NOvA uses, it probably won't be necessary to install a full Linux operating system (OS). Getting access to a terminal or a terminal emulator will probably be sufficient, since almost all work will be done remotely using the vm's. With that in mind, Mac users usually find that the built-in terminal meets their needs, while Windows users may be best served by a lightweight emulator like VirtualBox. Personally, I find it helpful to have a Linux installation available to me locally, so I use a dual-boot Windows/Ubuntu system. If you do choose to install Linux on a partition in your hard-drive, be very careful to backup your Windows data before beginning the installation. Editing the bootloader and partition table (without knowing what you're doing) is an easy way to render your machine unbootable.

Appendix B: Installing WimpSim for NOvA

Please note that there is a version of the WimpSim libraries (not the binaries themselves) installed in the $EXTERNALS area on the gpvm's, and that probably you should be using the runWimpSim standalone that is available in the EventGenerator package. Instructions on setting up WimpSim and running the standalone are available here.

Currently there is no working install of the WimpSim binaries available on the vm's. To produce one, you should follow these instructions[12]:

Instructions for installing the WimpSim library are available here. Unfortunately, following these instructions as written will not work on the gpvm's. For example, `ifort` is not installed on the vm's, so we will be using `gcc`.

Building pythia is straightforward if you want a standalone installation of WimpSim. If so, just download the source code from the previously linked WimpSim website, and do[13] `gcc -O -c pythia-6.4.26.f` in the desired install directory. This will produce a file called `pythia-6.4.26.o`.

If, however, you want to build an installation of WimpSim that can be used in conjunction with NOvA libraries (eg. runWimpSim), you will want to use the installation of pythia which is available as a ups product in the $EXTERNALS directory on the vm's (because this is the version that NOvA is linked against, and you cannot build WimpSim against a conflicting version). This version of pythia is setup when you call setup_nova, and you can see which version is currently active by doing `ups active | grep pythia` (as of the time of this writing, version 6.4.28d is active). You will want to know the location of the pythia object library that you will link against. The easiest way to do so is to do a `printenv | grep pythia` and to do a little searching until you find the .o file. If this is the goal, you will very likely need to write your own Makefile. You can use the example in the $EXTERNALS/wimpsim directory for guidance. As another note on compiling for use with NOvA code, you will need to include additional compiler flags: -fPIC and -malign-double. This applies to all the FORTRAN code that you will be compiling.

[12] These instructions were written while building in the development tag on 01/15/2016. Updates to the underlying software (like gfortran, for example) may render them defunct.
[13] The -O flag tells the compiler to optimize the built code for speed.

Extract the DarkSUSY archive into its own directory (`tar -zxf darksusy-5.1.3.tar.gz`). Then cd into the directory and do `./configure` and `make`. You will see some errors, but these can be safely ignored.

Extract the nusigma archive (`tar -zxf nusigma-1.17-pyr.tar.gz`). In the `src` directory, change the first line of the `makefile` to use gfortran. The pattern definitions at the bottom are not compatible with the version of gcc running on the vm's, so add these lines to the bottom of the makefile (caution - that should be a tab on the third line, not 8 spaces!):

```
.SUFFIXES : .o .f

%.o: %.f
	$(FC) $(FC_FLAGS) -I$(INC) $< -o $@
```

Once that is done, do `make`. The compilation will fail in the `mains` directory, but we don't need it, so don't worry about that.

Now extract wimpsim (`tar -zxf wimpsim-3.05.tar.gz`) and cd into the directory. Modify the makefile in the wimpsim root directory with the paths to the libraries we just compiled. You want the root directory for DarkSUSY and nusigma, not the lib directories. To make life easier for yourself, just use absolute paths and make sure you get the spelling correct.

Then you will have to modify four more makefiles (`wimpann/src/makefile`, `wimpann/mains/makefile`, `wimpevent/src/makefile`, and `wimpevent/mains/makefile`), setting the compiler to gfortran for each, and adding the pattern definition below to the two src and the sla-2.5.5 makefiles (note the small change):

```
.SUFFIXES : .o .f

%.o: %.f
	$(FC) $(FC_FLAGS) $(INC) $< -o $@
```

Additionally, in `wimpann/mains/makefile` and `wimpevent/mains/makefile,` remove the -lFH flag on line 43. The FeynHiggs library is not actually needed by WimpSim. Now add the following to `contrib/sla-2.5.5/makefile:`

```
%.o: %.F
	$(FC) $(FC_FLAGS) $(INC) -c  $< -o $@

%.o: %.f
	$(FC) $(FC_FLAGS) $(INC) -c  $< -o $@
```

Then just do a `make` in the wimpsim root directory, and everything should be built. The binaries will be in the `bin/` directory and the libraries in `lib/`.

As a final note, simply copying the installation to another directory won't necessarily work (in effect, you'll still be using the original directory), since the configuration script puts absolute paths to the build directory into header files that are then read at runtime. If you change the path to the installation, you will encounter errors at runtime. This also complicates the process of making a portable installation. To make the ups product, I found each spot in the code where the build path constants are used and rewrote it to use environment variables instead. Examine the differences between the code in $EXTERNALS/wimpsim and the released version to see what that entails.

Appendix C: Producing simulated events using the WimpSim binaries

Once installed, the two WimpSim binaries: `wimpann` and `wimpevent`, can be used to produce an estimation of the neutrino flux at the detector. Details on how this is done, and what are the inputs and outputs for each program, are summarized in the first few slides of this slidedeck and in the WimpSim documentation.

In order to produce simulated NOvA events using the WimpSim flux prediction, one must either:

- Use the outgoing particles (those marked with "O" in the `wimpevent` output file) as input to the NOvA detector simulation. This is the easiest method but does not tend to produce realistic events. The easiest interface to use in this case is the `TextFileGen` module. The comment block at the beginning of that file has some helpful instructions. Simply writing a script to read the `wimpevent` output text file and produce another text file is all that is needed for this method.
- Use the incoming neutrinos (marked with "I" in the `wimpevent` output) as input to GENIE, then use the GENIE output in the detector simulation. This is accomplished using the GENIEGen module.

The second option is more involved, since GENIE expects the neutrino flux to be provided in a particular format (namely, a GSimpleNtpFlux file). Originally, several ROOT scripts were written to accomplish this, which had to be run one at a time and which each read and then produced either a text or ROOT file. This is terribly inefficient, since the neutrino objects are represented in memory in WimpSim, so they never *need* to be written to disk in any form. The runWimpSim.cpp program takes advantage of this by linking against the WimpSim libraries to directly produce the GENIE-format flux with no need for intermediate text files or the WimpSim binaries themselves. Instructions for using it can be found in the WimpSim section in Part 4 above.

For the sake of completeness, here are some instructions for producing a GENIE flux file from WimpSim text output:

1. *Load the necessary GENIE libraries.* The GSimpleNtpFlux is a class defined in GENIE, and it depends on several of the GENIE packages. GENIE itself has additional dependencies, so a script that will produce GSimpleNtpFlux files needs to somehow have access to all of these libraries. Unfortunately, loading libraries interactively in ROOT is fairly limited in that libraries with circular dependencies cannot be loaded. This means you will have to combine all of the dependencies into a single library file which you can then load interactively in ROOT. Here are the steps to do so:

a. Run this command: `gcc -shared -o libmygenielib.so $GENIE/src/{Messenger,Algorithm,Utils,Numerical,Base,Registry,Interaction,FluxDrivers,EVGDrivers,EVGCore,GHEP,PDG}/*.o -I$LOG4CPP_INC -I`root-config --incdir` -L$LOG4CPP_FQ_DIR/lib -llog4cpp `root-config --glibs` -lEGPythia6`
b. Now add the necessary includes:
```
gInterpreter->AddIncludePath("/nusoft/app/externals/genie/v2_8_0i/Linux64bit+2.6-2.5-e6-debug/include/GENIE");
gInterpreter->AddIncludePath("/nusoft/app/externals/genie/v2_8_0i/Linux64bit+2.6-2.5-e6-debug/GENIE_R280/src/Utils");
```
c. Now load the libraries:
```
gSystem->Load("libTree.so");
gSystem->Load("libPhysics.so"); gSystem->Load("libxml2.so");
gSystem->Load("/<full path to file>/libmygenielib.so");
```
d. This process can be automated by adding the above lines to your `rootlogon.C` script. You will then want to update your `.rootrc` file in your home directory to point to the custom script.

2. *Run the script*. You can write your own script to take the `wimpevent` output and produce the flux ntuple, or you can use mine: `/nova/app/users/ram2aq/we2nova/runWimpSim.C` (not to be confused with the runWimpSim.cpp which is now part of the EventGenerator package). This ROOT macro must be run within a working installation of WimpSim (ie. the two files `wimpsim-3.05/bin/wimpann` and `wimpsim-3.05/bin/wimpevent` must exist). This is because it uses system calls to execute both binaries in turn, saving their output and processing it until it finally produces the desired ntuple format.

Once you have a GENIE-format flux file, you can use it as input to the GENIEGen module in the EventGenerator package, as discussed in the previous sections on simulating atmospheric neutrinos and WimpSim.

Appendix D: Outstanding Questions/Problems

Here is a brief summary of unanswered questions and incompletely addressed problems encountered so far in the project:

- At the trigger level and in the offline analysis, all cuts so far have been made using either track-level or slice-level information. More sophisticated techniques, like identifying vertices from neutrino interactions and Michel electrons (see Eric Culbertson's talk from the July 2015 collaboration meeting here), have not yet been adequately explored.
- It has become clear that most of the upward-going muons in the detector are likely caused by cosmic ray interactions nearby, rather than by atmospheric neutrinos (see Rob Mina's slides from July 2015 and January 2016). Since neither of the two upward-going muon triggers are cutting on elevation angle, it may be possible to improve our efficiency by implementing such a cut and loosening other requirements that are cutting into the efficiency for lower-energy muons.
- At the January 2016 collaboration meeting, Brian Rebel suggested using WindowTrack as the module for producing track objects for this analysis instead of KalmanTrack, which has been used so far.
- It was found early in the process of running reconstruction that jobs tended to fail on recent (<2 months) data files. Ultimately it was decided to use the tag S15-05-04b, with the Production/fcl/prod_reco_numi_job.fcl configuration file, and to use runs in the range 18398-19652 (exclusive). These decisions were made as a matter of convenience and because they seemed to work well. Since run 19651 occurred on May 26, 2015, several months of fairly continuous data-taking have already elapsed since the last piece of data was collected. Improvements in the reconstruction methods have likely also occurred in the meantime, so a more recent tag and an updated dataset should be used for future analyses.
- It seems that the hit-time resolution may not be the same in simulated events and in actual data. This changes the effects of the LLR and Chi2 cuts, and makes it difficult to estimate efficiency and purity for the cuts. As of the time of this writing, the effect is not well understood.
- As discussed in the atmospheric neutrino section, the method currently used to produce simulated atmospheric neutrino events is not quite ideal. The scripts should be replaced by a standalone C++ program (or, the GENIE atmospheric flux drivers should be used) and the method of choosing the point of origin for each neutrino should be better understood.
- By combining WimpSim or atmospheric flux predictions with GENIE, it is possible to assign to each simulated sample some exposure time or number of WIMP annihilations. How should an uncertainty be properly assessed on this exposure?

- The atmospheric flux predictions used thus far are for the MINOS experiment in Soudain, MN. The NOvA far detector is located fairly nearby in Ash River, MN. Is there a significant difference in the expected flux for these two locations?